COMSC-210 Lecture Topic 6
Linked Structures

Reference
Childs Ch. 7 (not 7.4.3)

Storing Values In Linked Structures
values stored in linked "nodes"
  using a `struct` inside the template

```
struct Node
{
  T value;
  bool inUse;
  Node* next;
};
```

"next" tracks location of next node
avoids need to reallocate and copy

The "start" And "end" Pointers
nodes link to each other
but what about the 1st node?
what if there are zero nodes?
...introducing the "start" pointer
  disembodied "next" pointer
  zero value means empty
data members for a linked list:

```
Node* start = 0;
Node* end = 0;
```

The "next" Pointer
`next =` {memory address of next node};
what about the last node's next?
  `next = 0;` // end marker

Adding One Node At End
create node, assign value, link in

```
Node* node = new Node;
node->inUse = false;
node->next = 0;
if (end) end->next = node;
else start = node;
end = node;
```

Adding n Nodes At End

```
for (i = 0; i < n; i++)
{
  Node* node = new Node;
  node->inUse = false;
  node->next = 0;
  if (end) end->next = node;
  else start = node;
  end = node;
}
```

Traversing A Linked List

A LinkedList Class Template
desired application -- works exactly like DynamicArray
`LinkedList<int> a(10);` // create linked list of 10 `int`s
public interface same as StaticArray (topic 3) and DynamicArray

Data Members
the start pointer: `Node* start;`
the end pointer: `Node* end;`
size tracker: `int siz;` // initially zero
capacity tracker: `int cap;`
set *default* initial capacity to zero -- not 100

Default Constructor
initialize `siz`
initialize `cap`
allocate `cap` nodes
set all `inUse` to false

`operator[]`, setter version
if key < 0, return dummy
if key >= cap, allocate new nodes with inUse set to false
traverse to node at key
  if not inUse, set inUse true, increment siz
  return it's "value"

`operator[]`, getter version
if key < 0, return dummy
if key >= cap, return dummy
traverse to node at key
  if not inUse, return dummy
  else return it's "value"

`containsKey`
if key < 0, return false
if key >= cap, return false
traverse to node at key, return it's "inUse"

`deleteKey`
if key < 0, return
if key >= cap, return
traverse to node at key
  if inUse, set inUse false, decrement siz

`clear`
set all inUse to false for all nodes
set size to zero

`Copy Constructor`
traverse copied LinkedArray's nodes
 create new node
 copy LinkedArray's node's value and inUse
copy cap and siz

```
// read-only
for (const Node* p = start; p; p = p->next)
  if (p->...) ...

// mutating
for (Node* p = start; p; p = p->next)
  p->inUse = false;
```

Finding A Node For A Key

```
int i = 0;
Node* p; // may be const...
for (p = start; p; p = p->next, i++)
  if (i == key)
    break;
if (p) // then p points to matching node
```

Deallocating All Nodes

```
while (start)
{
  Node* p = start->next;
  delete start;
  start = p;
}
end = 0;
```

Inefficiencies -- To Be Addressed Later...

traversal: `[i]` has to count from node zero each time it's called
`int size() const` has to count from node zero each time it's called

Big Data Time Issues

pertaining to the DVC schedule lab assignments:
"duplicate checking" has to scan a list of 35k (average) values
every time a new record is read
instead of a 1-dimensional list, try:
list of objects: one per term
in each list, store a seen section number
...OR...
list of objects: one per section number
in each list, store a seen term
instead of scanning 35K term+section combos,
scan ~50 terms and ~1000 sections
...OR...
scan ??? section numbers and ~50 terms